Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science 289 (2012) 3–14

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

High Performance Static Analysis for Industry

Mark Bradley, Franck Cassez, Ansgar Fehnker, Thomas Given-Wilson and Ralf Huuck

NICTA
University of New South Wales
Sydney, Australia
firstname.lastname@nicta.com.au

Abstract

Static source code analysis for software bug detection has come a long way since its early beginnings as a compiler technology. However, with the introduction of more sophisticated algorithmic techniques, such as model checking and constraint solving, questions about performance are a major concern. In this work we present an empirical study of our industrial strength source code analysis tool Goanna that uses a model checking core for static analysis of C/C++ code. We present the core technology and abstraction mechanism with a focus on performance, as guided by experience from having analyzed millions of lines of code. In particular, we present results from our recent study within the NIST/DHS SAMATE program. The results show that, maybe surprisingly, formal verification techniques can be used successfully in practical industry applications scaling roughly linearly, even for millions of lines of code.

Keywords: Validation, verification, static analysis, model checking, tools, SAMATE, C/C++

1 Introduction

Software development cycles are a major competitive aspect in many market segments including mobile phone handsets, games, and consumer electronics. The obvious goal is to deliver software as fast as possible, as cheaply as possible, and at the highest quality possible. For these reasons, automation and tool support play an important role. One of the areas for a high potential of automation and cost saving is testing and debugging, where around 50% of all development costs are spent.

One of the fastest growing tool spaces for testing and debugging is in *static analysis*. Static analysis is a complementary approach to traditional testing techniques. Instead of executing the source code on test suites, static analysis operates on the code compute solutions of semantic equations and detect problems that may lead to crashes, memory leaks, security flaws, etc. Originating from compiler optimiza-

doi:10.1016/j.entcs.2012.11.002

tion [2] static analysis has developed into sophisticated tools for bug and security vulnerability detection [7,15,13].

In recent years new algorithmic techniques have been developed by the formal methods community and approaches like model checking, SAT solving, and abstraction refinement are increasingly used for software analysis [14,5,8]. While these technologies can provide powerful capabilities, they come with the stigma of not being practicable for real-life systems, as being slow, and not being scalable to large code bases. That is, these new technologies are suspected to lack in performance or have to sacrifice accuracy.

The adoption of these new techniques by industry is only possible if the following problems are addressed properly.

- Scalability and Efficiency: new tools must be able to analyze large code bases while being very fast so that they can be used while programming.
- Accuracy: code analysis is done on an abstraction of programs, which may result in *false positives*, i.e. bugs that are artifact of the abstraction rather than real bugs in the program. The ratio of false positives versus issued warnings should be kept low to give useful feedback to the programmer.
- Smooth integration with popular IDEs: Software development now heavily relies on the use of IDEs like Eclipse or Visual Studio. Moreover, in some areas like embedded systems development, particular compiler versions (or build processes) are used and a versatile tool should accommodate them.

In this work, we report on the source code analysis tool called *Goanna* that performs static analysis for large scale industrial C/C++. Goanna is a commercial tool that uses standard model checking and other formal analysis techniques at its core to detect many classes of software bugs and security issues automatically at software development time.

We specifically address the challenges cited above and we present some detailed results from our recent participation in the Software Assurance Metrics And Tool Evaluation (SAMATE) [1] program run by the National Institute of Standards and Technology (NIST) and the Department of Homeland Security (DHS). We present a number of qualitative results and real-life software bugs found in large open-source code bases. Additionally we give detailed quantitative analysis on the scalability of our model checking approach, both in terms of lines of code (LoC) as well as number of checks performed.

The rest of this paper is organized as follows. In Section 2 we motivate the use of static analysis in the software development life-cycle. Section 3 examines the techniques used by Goanna to achieve scalable and accurate results. Section 4 presents results achieved in SAMATE tests. Section 5 discusses our experience with integrating Goanna into existing development environments. Finally, Section 6 draws conclusions and discusses future work.

[1] http://samate.nist.gov/

2 Static Analysis in Software Engineering

To better understand the role of static analysis in industrial software development, we briefly characterize where static analysis, and in essence our Goanna tool, fits into the software development life-cycle (SDLC) and show how it integrates with existing software engineering processes in practice.

In a simplified water flow model of the software development life-cycle there are four stages: design, implementation, quality assurance, and finally the released product. While most software development processes are not executed in this strict sequential order, it nonetheless highlights an important fact: *the greater the time between when a software bug introduced into a system and its point of detection, the greater the cost of fixing that bug.* For instance, a bug introduced in the design stage is rectified at relatively low costs if it is detected immediately in the same stage. However, it is often prohibitively expensive when detecting a serious design flaw after product release.

Implementation bugs are often even harder to detect than design bugs. This is why around 50% of all software costs are spend in testing and debugging. Consequently, any automated support that can identify certain classes of bugs earlier on leads to immediate cost savings. Static program analysis is such an automated technology working directly at development time on the source code. Unlike traditional testing techniques, static analysis does not require manual intervention or a completed build, but can be applied to sub-components or single files.

There is a large cost benefit on cutting down traditional implement-test-debug cycles. It is worth noting that static analysis is not a replacement for traditional testing, but a complementary technique for many non-functional requirements, such as detecting memory leaks, crashes and security holes.

3 Goanna Core Technology

In this section we describe the underlying core technology of Goanna. In particular, we explain the model checking approach to static analysis, core abstract interpretation applications, and additional techniques for path-sensitive and inter-procedural analysis. The combination of these technologies is used to detect potential crash causing code, security vulnerabilities, memory leaks and the like. A detailed list of detected vulnerabilities and checks that Goanna can perform can be found in [12].

3.1 Model Checking

Some of the checks performed by Goanna are beyond the scope of syntactical analysis or variable range analysis as they are path-sensitive. This is why one core technology implemented in Goanna is *model checking*. A model is a transition system annotated with *atomic propositions*. A check or property is defined in a *temporal logic* [3] that can capture path-dependent specifications.

Model checking [3] is a technique that enables one to check whether all paths in a model satisfy a property. If the property is violated, a counter-example is provided

```
   void foo() {
0:  int x, *a;
1:  int *p=malloc(sizeof(int));
2:  for(x=10;x>=0;--x) {
3:    a=p;
4:    if (x<=1)
5:     free(p);
6:  }
7: }
```

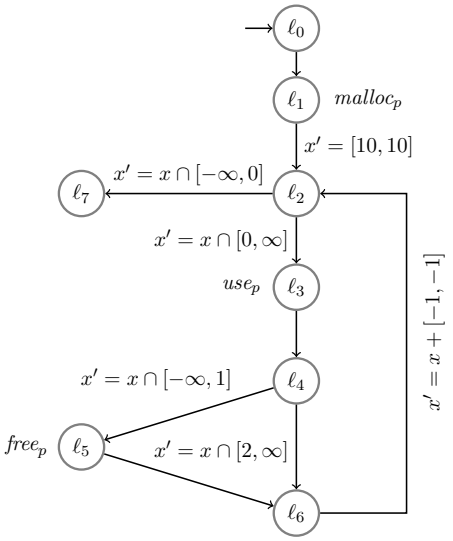


Fig. 1. Example program and the generated CFG with annotations, and generated interval equations for variable x. The annotated CFG is used for model checking, the interval equations for abstract data tracking.

by the model checking algorithm.

We apply model checking to source code by mapping a C/C++ program to its corresponding control flow graph (CFG), and adding labels to the CFG which are the atomic propositions of interest (see [9]).

Consider the contrived program `foo` in Fig. 1. The CFG of `foo` is annotated with atomic propositions [2] describing the operations performed on pointer p. The labels are $malloc_p$, use_p and $free_p$ which respectively stands for p is allocated some memory, p is used, and p is freed. An important property of the program should be that the pointer p is not freed and then used. Such a property can be expressed in Computation Tree Logic (CTL):

$$AG\ (malloc_p\ implies\ AG\ (free_p\ implies\ \ not(EF\ use_p))),$$

where AG stands for "for all paths and in all states" and EF for "there exists a path and there exists a state". This CTL formula means that whenever there is a `malloc` for a resource p, if it is followed by a `free` of p, then there is no path such that p is used later on.

The model checking technique has many advantages among them: (1) fine-tuning properties is easy and we can express stronger/weaker requirements by changing the CTL path quantifier, i.e. changing an A to an E and visa versa; (2) unlike standard static analysis, the model checker produces a counter-example [3] when a property is violated and this can be fed back to the programmer.

Implementation. When analyzing source code, the models are usually small but the properties to check are numerous (see Section 4.2). Most of the CTL formulas do not have nested modalities and thus are rather easy to check. This is why Goanna

[2] How the labelling is carried out is described in Section 3.2.

[3] This counter-example on the CFG might be spurious though.

features an explicit state model checker that is specifically designed to efficiently check a large number of properties on a model, reusing previously computed results (by caching).

3.2 Tree-Pattern Matching

A prerequisite to model checking is to annotate the source code (or CFG) with the atomic propositions of interest. In our approach we apply pattern matching to the internal representation of the parsed source code, i.e. the *abstract syntax tree* (AST).

Implementation. Basic regular expression pattern matching is sufficient for some queries, such as whether a library function such as `strcpy` is used or not. However, queries can become more complicated, and a series of interdependent queries may be used for more advanced checks, for example to identify inconsistent use of semantic attributes. For this purpose Goanna uses tree-pattern matching [4] on the AST. This enables expressive queries taking branching substructures into account and is flexible enough to define a wide range of non path-dependent checks.

3.3 Abstract Data Tracking

The techniques introduced so far are well suited for path-dependent checks. However, in order to detect arithmetic errors e.g. division by zero or buffer overflows, the ranges of values for certain variables must be considered.

A general technique to automatically approximate and track data values is *abstract interpretation* [6]. Goanna implements abstract interpretation to estimate the potential ranges for each interger and pointer variable at each program location. This enables us to, for instance, estimate the potential index values whenever an array is accessed or NULL pointer dereference.

Implementation. Different domains can be used for data tracking with varying levels of precision. Goanna uses a variant of interval constraint analysis that is reasonably precise while remaining fast. For this purpose we translate the relevant program semantics into equations over intervals as seen in Fig. 1. Here, intervals $[a, b]$ are represented by their lower bound a and upper bound b, which can be sometimes unknown in which case they are $\pm\infty$. Operations on the intervals are defined in terms of addition, multiplication, union, and intersection. Each equation constraint involves a variable with its new value denoted as its primed version. The equations and algorithms implemented in Goanna are based on some efficient techniques described in [11].

3.4 Inter-Procedural Analysis

While the aforementioned techniques help to detect various vulnerabilities and are often complementary, one of the key factors to success is being able to scale to large code bases while remaining accurate. Many potential source code bugs require understanding of the overall call structure of a program, and being able to track

data and control flow across function boundaries.

Implementation. To overcome this challenge we have developed a compositional approach computing *function summaries* automatically. These function summaries contain information that is needed for particular checks e.g. ranges of variables returned by the function, existence of path to NULL pointer dereference. Instead of propagating information of the whole function, which can be prohibitively large, only the summary information is used. The information for each function is stored in a database that can be enriched when some new knowledge is available e.g. constraints on the input data of the function.

3.5 Data Flow Analysis

On top of the previous techniques we also use data flow analysis [2] to examine the flow of information between variables and other elements of interest. An example is checking for the flow of tainted data in a security context. While data flow analysis is useful to track information flow, it is less amendable to precision improving techniques such as refinement and also does not typically return any counter-example traces if a program property is violated.

3.6 Current Limitations

As most of the static analysis tools, Goanna is not *sound* and thus it cannot be guaranteed that every defect will be discovered. Moreover, the commercial version of Goanna currently does not handle: pointer aliasing, multi-theading and *refinement*.

4 Performance Results

We report on Goanna's performances based on the independent evaluation project SAMATE by NIST and DHS. The project conducted the third Static Analysis Tool Exposition (SATE) in 2010 to evaluate static analysis tools and try and find security bugs in source C/C++ code [1].

Various test suites were provided for SATE 2010 among them the code of open source projects *Dovecot*, *Wireshark*, and *Chromium*. A detailed report with the weaknesses discovered by each tool was submitted by each group to NIST experts. The experts analyzed the results and assigned one of the following categories to each discovered weakness.

- Security: a weakness relevant to security.
- Quality: a weakness pertaining to poor code quality. The weakness may not be relevant to security, but requires developer attention.
- Insignificant: a true but insignificant weakness.
- False: Not a weakness, i.e. a false positive.
- Unknown: unable to determine correctness or significance.

NIST experts more specifically analyzed the reports of the different tools for

Dovecot. The summary of this analysis is available for download under the section *Sate Analysis/Analysis Subset* [4] .

The figures we report hereafter were obtained using Goanna Central 2.5 (the command line version) with the default settings and whole-program analysis. The experiments were conducted on a Ubuntu 10.11 Virtual Machine with 2GB memory, on a Windows 7 host, running on an Intel Core i5 2.5 GHz processor. While this setup is not optimized for speed, the quantitative results are indicative of Goanna's performance.

4.1 The Dovecot Code Base

The Dovecot code base consists of 672 files and each file has on average 322 LoC before pre-processing, and 2685 LoC after pre-processing. The maximum number of LoC is 2685 before pre-processing and 6623 after pre-processing. Each file consists on average of 21 separate functions with a maximum of 128 functions per file. Fig. 2

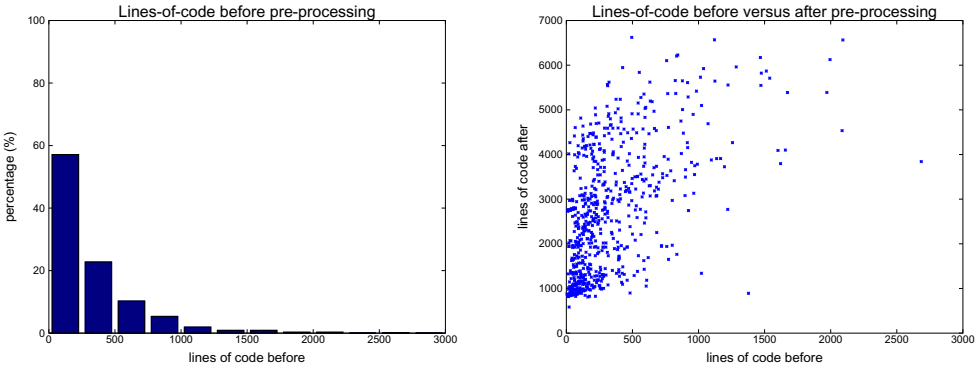


Fig. 2. Lines of code distribution per file and before pre-processing versus after pre-processing.

(left) shows the distribution of the LoC before pre-processing; nearly 60% of all files have less than 250 LoC. The number of functions, and LoC after pre-processing are similarly distributed, with about 50% smaller than the mean.

Fig. 2 (right) shows that there is only a weak correlation between the LoC before and after pre-processing, with quite a few outliers. There are a fair number of files that have few LoC before, but a rather large number of LoC after pre-processing.

Dovecot is a good representative of a large size software project (this is why it was chosen by NIST) and the results we give in the sequel are inline with what is obtained on similar industrial code bases.

4.2 Scalability and Runtime

As described in Section 3, Goanna uses model checking for the analysis of C/C++ source code. It model checks the program labelled CFGs against CTL formulas. A classic result is that CTL model checking is linear in the number of states and the

[4] http://samate.nist.gov/SATE2010/resources/sate2010.tgz

size of the CTL properties. Each Goanna check translates to one CTL property. Moreover, for each program variable there might be one check of the same class. For example, each variable has to be checked for being uninitialized. The scalability of our analysis will therefore depend on the size of CFG and number of properties that have to be checked.

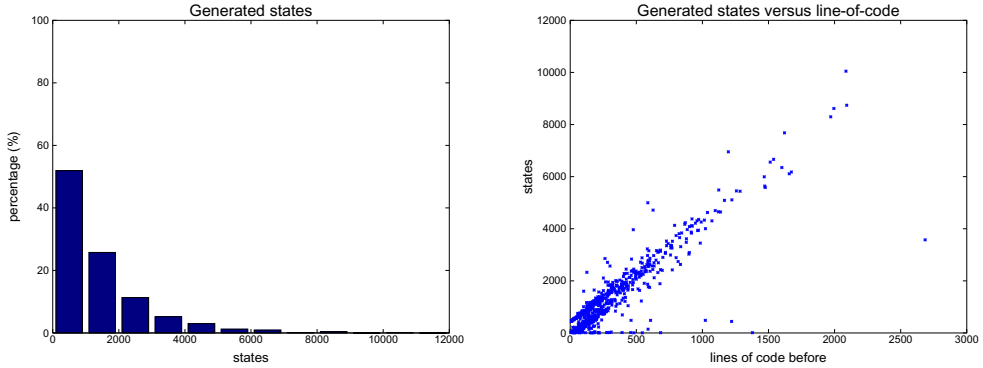


Fig. 3. Number of states per file and runtime per file

Fig. 3 (left) shows the distribution of the sizes (number of states of the CFG) of the CFGs per file. Each file has on average 1372 states, but at most 10048. Fig. 3 (right) shows that the number of states increases roughly linearly with the number of LoC before pre-processing.

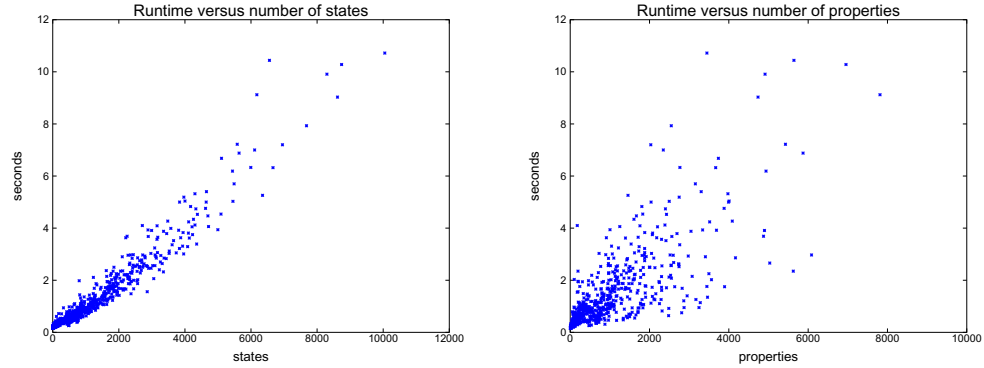


Fig. 4. Runtime versus number of states and properties per file.

Fig. 4 depicts the relation between the runtime and both the number of states and the number of CTL properties [5]. If we look at the relation between runtime and the number of generated states we see that there is an almost linear relation between the two. Since the number of states depends linearly on the LoC (Fig. 3) this also means, the runtime is linear in the size of the code base. There is a less pronounced linear relation between the number of properties and the runtime. Indeed, this relation seems to be dominated by the relation between the number

[5] We omit one outlier with 20636 properties and 3125 states.

of properties and the number of states. For Dovecot 27% of files the Goanna tool generates more properties than states.

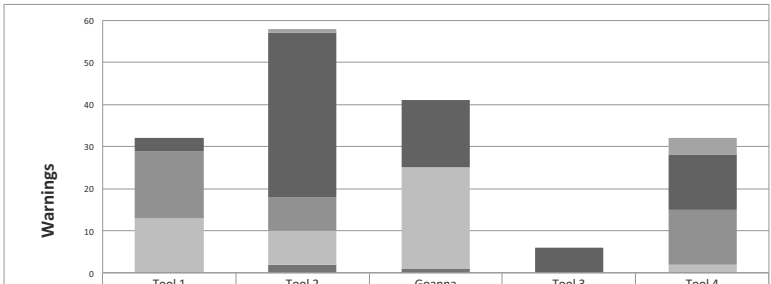
The runtime per file is on average 1.4 seconds, well in the order of compile time. Recall that the average size after pre-processing is 2685 LoC. As Fig. 4 (left) shows more than 50% of files are analysed in less than 1 second.

In summary, the code size and in turn the number of states that are generated is a good indicator for the runtime. It is worth mentioning though, that model checking itself only accounts for around 20% of the overall runtime. The rest of the runtime is spent in parsing, generating the models, pattern matching, or analyzing data flow. The number of generated states is also a good measure for the complexity of the source code, and thus for the complexity of other types of analysis techniques.

4.3 Quality and Accuracy of the Analysis

Fig. 5 gives the number of weaknesses (for each category defined at the beginning of this section) detected by some tools that took part in the SATE 2010 evaluation for Dovecot. [6]

Goanna is rather accurate and has the highest number (25) true weaknesses (security and quality) compared to 12, 10 and 2 for the other tools: this shows that model-checking can actually improve the accuracy of a tool.



	Tool 1	Tool 2	Goanna	Tool 3	Tool 4
Unknown	0	1	0	0	4
"false"	3	39	16	6	13
Insignificant	16	8	0	0	13
Quality	13	8	24	0	2
Security	0	2	1	0	0

Fig. 5. Distribution of the weaknesses

Fig. 6 allows us to determine the percentage of false positives, which is a good measure of the quality of the checks. To define false positives, the NIST experts determined two categories of weaknesses: *Security* and *Quality* ones are *Good* weaknesses in the sense that identify real issues; the other category contains the *Insignificant* and *False* weaknesses that are *Bad* as they correspond to false positives or trivial issues. Fig. 6 shows the ratio Good/Bad for each tool. Goanna performs the very well with less than 40% false positives.

[6] No general conclusion about the tools can de drawn as this experiment concerns on project Dovecot.

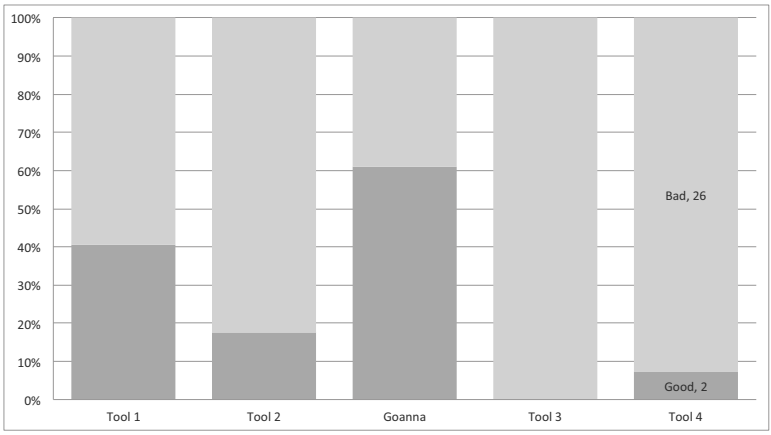


Fig. 6. False Positives

5 Integration

All these results are great, but ease-of-use is also vitally important to success in industry. Here we comment briefly on our experience with integrating Goanna into industrial environments and systems.

Most important for industrial tool adoption is a seamless integration into existing development processes, as process changes are often hard to achieve and costly to implement. Therefore, a successful uptake of static analysis tools requires easy integration into the SDLC.

Early static analysis tools for C/C++ required a complex setup procedure: the location of header files needed to be defined explicitly, compiler configurations had to be meticulously set up, and analyzers were not integrated with standard IDEs. Further, code annotations were sometimes required. All these steps can be serious hurdles and hard to maintain in the long-term.

Modern static analysis tools overcome this in various ways: they can either record the build process and play it back to the analysis tool later, or directly integrate just like a compiler into Makefiles and the like. Our Goanna tool is available as both an IDE and server-targeted (command line) version. The IDE variant of Goanna integrates with Eclipse as well as Microsoft Visual Studio (2005, 2008, 2010 & 11 beta). These variants can be installed via a "click-through" installer. The setup and configuration is done completely by familiar check boxes in the IDE. The command line variant can be used exactly like a compiler, e.g. instead of calling the compiler `gcc` the developer calls our analysis tool `goannacc`. The tool itself will automatically identify includes, compiler switches and so on. Moreover, a fine-grained setup can be achieved through command line options or text configuration files. As a result, for either variant of Goanna there are minimal changes in the software development process and in most cases the software developer never has to switch windows between his normal environment and the analysis tool.

6 Summary and Future Directions

In this work we presented our practical results from using automated formal methods, in particular model checking, for static bug detection of industrial software. An important contribution is the empirical demonstration that our model checking approach to static analysis scales roughly linearly to millions of lines of C/C++ code. Moreover, we have shown that our Goanna tool is able to detect previously unknown and relevant safety and security flaws automatically in large C/C++ code bases.

To some extend it might be surprising that model checking technology can indeed scale close to linearly for real-life C/C++ software, given that the *state explosion problem* is one of the most common phenomena associated with model checking. However, there are three significant explanations for this. Firstly, the core issue of state explosion typically comes from having *concurrent* systems with an inevitable exponential number of executions. We are, however, considering sequential C/C++ code. Secondly, CTL model checking itself has a worst-case complexity that is linear in the size of the model and linear in the size of the CTL formula. Hence, our results are perfectly in line with this. Thirdly, our abstractions are on such a level, where they create very manageable state spaces. This is particularly true for our summary based inter-procedural approach.

Future work is to push the envelope further. This means, we endeavor to successively add more formal verification techniques such as SMT solving and automated theorem proving to create an even more fine-grained analysis without overly compromising performance. Some early results in that direction can be found in [10]. Moreover, we see another important challenge in addressing real concurrency issues resulting from multi-threaded code.

References

[1] *Report on the Third Static Analysis Tool Exposition (SATE 2010)* (2011).
URL http://samate.nist.gov/docs/NIST_Special_Publication_500-283.pdf

[2] Aho, A. V., R. Sethi and J. D. Ullman, "Compilers: Principles, Techniques and Tools," Addison-Wesley, 1986.

[3] Baier, C. and J.-P. Katoen, "Principles of Model Checking (Representation and Mind Series)," The MIT Press, 2008.

[4] Benedikt, M., W. Fan and G. M. Kuper, *Structural properties of xpath fragments*, in: *ICDT '03: Proceedings of the 9th International Conference on Database Theory* (2002), pp. 79–95.

[5] Clarke, E., D. Kroening, N. Sharygina and K. Yorav, *SATABS: SAT-based predicate abstraction for ANSI-C*, in: *Proc. TACAS 2005*, LNCS 3440, 2005, pp. 570–574.

[6] Cousot, P., *Semantic foundations of program analysis*, in: S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, Prentice-Hall, Inc., New Jersey, 1981 pp. 303–342.

[7] Coverity, *Prevent for C and C++*.
URL http://www.coverity.com

[8] D'Silva, V., D. Kroening and G. Weissenbacher, *A survey of automated techniques for formal software verification*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) **27** (2008), pp. 1165–1178.

[9] Fehnker, A., R. Huuck, P. Jayet, M. Lussenburg and F. Rauch, *Model checking software at compile time*, in: *Proc. TASE 2007* (2007).

[10] Fehnker, A., R. Huuck and S. Seefried, *Counterexample guided path reduction for static program analysis*, in: *Concurrency, Compositionality, and Correctness*, Lecture Notes in Computer Science **5930** (2010), pp. 322–341.

[11] Gawlitza, T. and H. Seidl, *Precise fixpoint computation through strategy iteration.*, in: *ESOP*, LNCS 4421 (2007), pp. 300–315.

[12] Goanna, *Goanna Reference Guide*.
URL http://archive.redlizards.com/docs/CommandLineReferenceGuide.pdf

[13] GrammaTech, *CodeSurfer*.
URL http://www.grammatech.com/

[14] Holzmann, G., *Static source code checking for user-defined properties*, in: *Proc. IDPT 2002*, Pasadena, CA, USA, 2002.

[15] Klocwork, *Klocwork Insight*.
URL http://www.klocwork.com/